Dynamic Memory Allocation

First Edition

January 1988

Richard M. Stallman

Free Software Foundation

Copyright © 1988 Richard M. Stallman.

Published by the Free Software Foundation (675 Mass Ave, Cambridge MA 02139). Printed copies are available for $10 each.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions.

Table of Contents

1. Dynamic Memory Allocation 1
1.1 Explanation of Dynamic Memory Allocation 1
1.2 Dynamic Allocation and C 1
1.3 `malloc`: Unconstrained Allocation 2
1.3.1 Allocation with `malloc` 2
1.3.2 Freeing Memory Obtained with `malloc` 4
1.3.3 Changing the Size of a Block 4
1.3.4 Allocating Cleared Space 5
1.3.5 Efficient Use of `malloc` 6
1.3.6 Allocation with Specified Alignment 6
1.3.7 Warnings when Low on Memory 7
1.3.8 `malloc` and Signal Handlers 8
1.3.9 Summary of `malloc` and Related Functions 8
1.4 Obstacks 9
1.4.1 Preparing to Use Obstacks 9
1.4.2 Obstack Functions and Macros 10
1.4.3 Obstack Data Structure 11
1.4.4 Allocation in an Obstack 12
1.4.5 Freeing Objects in an Obstack 13
1.4.6 Growing Objects 13
1.4.7 Extra Fast Growing Objects 15
1.4.8 Status of an Obstack 16
1.4.9 Alignment of Data in Obstacks 16
1.4.10 Obstack Chunks 17
1.4.11 Obstacks and Signal Handling 18
1.4.12 Summary of Obstack Function/Macros 19
1.5 Automatic Storage with Variable Size 20
1.5.1 Example of `alloca` 21
1.5.2 Advantages of `alloca` 22
1.5.3 Disadvantages of `alloca` 22
1.5.4 GNU C Variable-Size Arrays 23

1. Dynamic Memory Allocation

The GNU system provides three different methods for allocating memory space under explicit program control.

- The `malloc` facility allows fully general dynamic allocation. See section 1.3 [Malloc], page 2.
- Obstacks are another facility, less general than malloc but more efficient and convenient. See section 1.4 [Obstacks], page 9.
- The function `alloca` lets you allocate storage dynamically that will be freed automatically. See section 1.5 [Alloca], page 20.

1.1 Explanation of Dynamic Memory Allocation

Dynamic memory allocation is a technique in which programs determine as they are running where to store some information.

You need dynamic allocation when the number of memory blocks you need, or how long you continue to need them, depends on the data you are working on.

For example, you may need a block to store a line read from an input file; since there is no limit to how long a line can be, you must allocate the storage dynamically and make it dynamically larger as you read more of the line.

Or, you may need a block for each record or each definition in the input data; since you can't know in advance how many there will be, you must allocate a new block for each record or definition as you read it.

When you use dynamic allocation, the allocation of a block of memory is an action that the program requests explicitly. You call a function or macro when you want to allocate space, and specify the size with an argument. If you want to free the space, you do so by calling another function or macro. You can do these things whenever you want, as often as you want.

1.2 Dynamic Allocation and C

The C language itself supports two kinds of allocation:

- *Static allocation* is what happens when you declare a static variable. Each static variable defines one block of space, of a fixed size. The space is allocated once, when your program is started, and is never freed.
- *Automatic allocation* happens when you declare an automatic variable, such as a function argument or a local variable. The space for an automatic variable is allocated when the compound statement containing the declaration is entered, and is freed when that compound statement is exited.

  In GNU C, the length of the automatic storage can be an expression that varies. In other C implementations, it must be a constant.

You use dynamic allocation in C if neither static nor automatic allocation will serve.

C does not define a storage class "dynamic", so the only way to refer to dynamically allocated space is through pointers. If you want to allocate dynamically some space to hold a `struct foobar`, you cannot declare a variable of type `struct foobar` whose contents are the dynamically allocated space. But you can declare a variable of pointer type `struct foobar *` and assign it the address of the space. Then you can use the operators '`*`' and '`->`' on this pointer variable to refer to the contents of the space.

1.3 `malloc`: Unconstrained Allocation

The most general dynamic allocation facility is `malloc`. It allows you to allocate blocks of memory of any size at any time, make them bigger or smaller at any time, and free the blocks in any order (or never).

1.3.1 Allocation with `malloc`

To allocate a block of memory, call `malloc`. The ANSI C declaration of `malloc` looks like this:

```
void * malloc (int size);
```

The argument *size* is the length of the block you want, measured in bytes.

The `malloc` function returns a pointer to the beginning of the block. The contents of the block are undefined; you must initialize it yourself (or use `calloc` instead; see section 1.3.4 [Cleared

Space], page 5). Normally you would cast the value as a pointer to the kind of object that you want to store in the block, like this:

```
struct foo *ptr;
...
ptr = (struct foo *) malloc (sizeof (struct foo));
if (ptr == 0) abort ();
bzero (ptr, sizeof (struct foo));
```

When allocating space for a string, the argument to `malloc` must be one plus the length of the string. This is because a string is terminated with a null character that doesn't count in the "length" of the string but does need space. For example:

```
char *ptr;
...
ptr = (char *) malloc (length + 1);
```

If no more space is available, `malloc` returns zero. You should check the value of *every* call to `malloc`. It is useful to write a subroutine that calls `malloc` and reports an error if the value is zero, returning only if the value is nonzero. This function is conventionally called `xmalloc`. Here it is:

```
void *
xmalloc (int size)
{
  register void *value = malloc (size);
  if (value == 0)
    fatal ("virtual memory exhausted");
  return value;
}
```

Here is a real example of using `malloc` (by way of `xmalloc`). The function `savestring` will copy a sequence of characters into a newly allocated null-terminated string:

```
char *
savestring (char *ptr, int len)
{
  register char *value = (char *) xmalloc (len + 1);
  bcopy (ptr, value, len);
  value[len] = 0;
  return value;
}
```

The address of the block that `malloc` gives you is always a multiple of eight, which is round

enough to be valid for any type of data. Only rarely is any higher boundary (such as a page boundary) necessary; for those cases, use `memalign` or `valloc` (see section 1.3.6 [Aligned Space], page 6).

Note that the memory located after the end of the block is likely to be in use for something else; perhaps a block already allocated by another call to `malloc`. If you attempt to treat the block as longer than you asked for it to be, you are liable to destroy the data that `malloc` uses to keep track of its blocks, or you may destroy the contents of another block. If you have already allocated a block and discover you want it to be bigger, use `realloc` (see section 1.3.3 [Changing Size], page 4).

1.3.2 Freeing Memory Obtained with `malloc`

When you no longer need a block that you got with `malloc`, use the function `free` to make the block available to be allocated again. This is how `free` is declared in ANSI C:

```
void free (void *ptr);
```

Freeing a block alters the contents of the block. **Do not expect to find any data (such as a pointer to the next block in a chain of blocks) in the block after freeing it.** Copy whatever you need out of the block before freeing it!

You cannot reduce the total memory space used by the program by calling `free`, because it does not currently know how to return the memory to the operating system. The purpose of calling `free` is to allow a later later call to `malloc` to reuse the space. In the mean time, the space remains in your program as part of a free-list used internally by `malloc`.

Therefore, there is no point in freeing blocks at the end of a program, when you are not going to allocate any more.

1.3.3 Changing the Size of a Block

Often you do not know for certain how big a block you will ultimately need at the time you must begin to use the block. For example, the block might be a buffer that you use to hold a line being read from a file; no matter how long you make the buffer initially, you may encounter a line that is longer.

You can make the block longer by calling `realloc`. Here is the declaration of `realloc` in ANSI C:

```
void * realloc (void *ptr, int newsize);
```

The argument *ptr* is the address of the existing block and *newsize* is the new size you want.

Since the space after the end of the block may be in use, `realloc` may find it necessary to copy the block to a new address where more free space is available. The value of `realloc` is the new address of the block. If the block needs to be moved, `realloc` copies the old contents.

Like `malloc`, `realloc` may return zero if no memory space is available to make the block bigger. When this happens, the GNU version of `realloc` is guaranteed not to alter the original block; some Unix versions of `realloc` may destroy its contents.

In most cases it makes no difference what happens to the original block when `realloc` fails, because the application program cannot continue when it is out of memory, and the only thing to do is to give a fatal error message. Often it is convenient to write and use a subroutine, conventionally called `xrealloc`, that takes care of the error message as `xmalloc` does for `malloc`:

```
void *
xrealloc (void *ptr, int size)
{
  register void *value = realloc (ptr, size);
  if (value == 0)
    fatal ("Virtual memory exhausted.");
  return value;
}
```

You can also use `realloc` to make a block smaller. The reason you would do this is to avoid tying up a lot of memory space when only a little is needed. Making a block smaller sometimes necessitates copying it, so it can fail if no other space is available.

If the new size you specify is the same as the old size, `realloc` is guaranteed to change nothing and return the same address that you gave.

1.3.4 Allocating Cleared Space

The function `calloc` allocates memory and clears it to zero. It is declared thus in ANSI C:

```
void * calloc (int nelts, int eltsize);
```

The block is made long enough to contain a vector of *nelts* elements, each of size *eltsize*, and its contents are cleared to zero before `calloc` returns. You could define `calloc` as follows:

```
void *
calloc (int nelts, int eltsize)
{
  int size = nelts * eltsize;
  void *value = malloc (size);
  if (value != 0)
    bzero (value, size);
  return value;
}
```

People rarely use `calloc`, because it is equivalent to such a simple combination of other features that are more often used. It is a historical holdover that is not quite obsolete.

1.3.5 Efficient Use of `malloc`

To make the best use of `malloc`, it helps to know that `malloc` always dispenses memory in units of powers of two. It keeps separate pools for each power of two. In addition, it needs four bytes before the beginning of each block for record-keeping.

Therefore, when you call `malloc` with argument n, the amount of space actually used up is the smallest power of two not less than $n + 4$. If you are free to choose the size of a block in order to make `malloc` more efficient, make it four less than a power of two.

In addition, the pools for different powers of two remain separate forever; a block of 28 bytes (32 minus 4) can never be split into two blocks of 12 bytes (16 minus 4). So it is good if you manage to use blocks of the same size for as many different purposes as possible.

1.3.6 Allocation with Specified Alignment

The address of a block returned by `malloc` or `realloc` is always a multiple of eight. If you need a block whose address is a multiple of a higher power of two than that, use `memalign` or `valloc`, declared as follows in ANSI C:

```
void * memalign (int size, int boundary);
void * valloc (int size);
```

`memalign` allocates a block of *size* bytes whose address is a multiple of *boundary*. *boundary* must be a power of two! It works by calling `malloc` to allocate a somewhat larger block, and then returning an address within the block that is on the desired boundary.

Using `valloc` is like using `memalign` and passing the page size as the value of the second argument. Its advantage is that you don't need to determine the page size explicitly (something which cannot be done portably on Unix systems).

1.3.7 Warnings when Low on Memory

When you call `malloc` and no more memory is available, it returns zero, which allows you to report an error. But in interactive programs it is often useful to give the user a warning when memory is nearly full but not quite. To request warnings, call `malloc_init`, declared as follows in ANSI C:

```
void malloc_init (void *base_ptr,
                  void (*warn_function) (const char *));
```

Here *warn_function* should be a function defined by you; `malloc` will call this function, with a constant string as argument, when it is time to warn the user. There will be three warnings, first when 75% of available memory is used, then when 85% is used, and finally when 95% is used. The string passed to *warn_function* tells the user how much of the memory is in use.

There is no guarantee that you cannot run out of memory instantly if you have not yet received a warning. On the contrary, you can always run out right away if you ask for one very large block. But as things normally go, the warnings are a good thing to tell the user.

Normally, you should supply zero for the argument *base_ptr*. This argument is used to tell `malloc` the address at which impure data starts. This value enables `malloc` to interpret the information from the system about how much more space is available. But normally `malloc` can get this information without help, because it is the address of the symbol `etext`. Zero means to use this value for the *base_ptr*. The only occasion to pass a nonzero value is when you have altered the executable file or your memory image in some way so that the address of `etext` no longer indicates where impure data starts.

The function `malloc_init` is a GNU extension not available unless you use the GNU version of `malloc`.

1.3.8 `malloc` and Signal Handlers

It is safe to call the `malloc` from a signal handler if you are using the GNU implementation of `malloc`. So may the other pure-allocation functions `calloc`, `valloc` and `memalign`.

But this is not a standard feature; most other implementations of `malloc` are not safe to call from signal handlers.

It is not safe to call `free` from a signal handler. Because `realloc` can use `free`, it is not safe to call `realloc` either. If you were to do this, serious unreproduceable bugs would occur if ever a signal arrives while `malloc` is running and `free` is called while `malloc` is half-done.

1.3.9 Summary of `malloc` and Related Functions

Here is a summary of the functions that work with `malloc`:

`malloc` (*size*)
: Allocate a block of *size* bytes. See section 1.3.1 [Allocation], page 2.

`free` (*addr*)
: Free a block previously allocated by `malloc`. See section 1.3.2 [Freeing], page 4.

`realloc` (*addr*, *size*)
: Make a block previously allocated by `malloc` larger or smaller, possibly by copying it to a new location. See section 1.3.3 [Changing Size], page 4.

`calloc` (*nelts*, *eltsize*)
: Allocate a block of *nelts* * *eltsize* bytes using `malloc`, and zero its contents. See section 1.3.4 [Cleared Space], page 5.

`valloc` (*size*)
: Allocate a block *size* bytes, starting on a page boundary. See section 1.3.6 [Aligned Space], page 6.

`memalign` (*size*, *boundary*)
: Allocate a block *size* bytes, starting on an address that is a multiple of *boundary*. See section 1.3.6 [Aligned Space], page 6.

`malloc_init` (*base_ptr*, *warn_function*)

Request warnings when memory is 75% allocated, 85% allocated, and 95% allocated. Warnings are given by calling *warn_function*. See section 1.3.7 [Warnings], page 7.

1.4 Obstacks

An *obstack* is a pool of memory containing a stack of objects. Within each obstack, the last object allocated must always be the first one freed, but distinct obstacks are independent of each other.

Aside from this one constraint of order of freeing, obstacks are totally general: an obstack can contain any number of objects of any size. They are implemented with macros, so allocation is usually very fast as long as the objects are usually small. And the only space overhead per object is the padding needed to start each object on a suitable boundary.

The objects in an obstack are packed into large blocks called *chunks*, which also form a stack. The obstack adds a new top chunk to the stack whenever you allocate an object that won't fit in the previous top chunk.

It is up to you to tell obstacks where to get the chunks. Usually they are obtained using `malloc` directly or indirectly.

1.4.1 Preparing to Use Obstacks

To use obstacks, you must first include the header file '`obstack.h`', like this:

```
#include "obstack.h"
```

You must also define two functions or macros that will be called by the obstack package. One, `obstack_chunk_alloc`, is used to allocate the chunks (large blocks) of memory into which objects are packed. The other, `obstack_chunk_free`, is used to return chunks when the objects in them are freed. You must define or declare them in each source file that uses `obstack_init` (see section 1.4.3 [Structure], page 11).

Usually these are defined to use `malloc` via the intermediary `xmalloc` (see section 1.3.1 [Allocation], page 2). Though the storage you get using obstacks really comes from `malloc`, using

obstacks is faster because `malloc` is called less often, for larger blocks of memory. Here are the macro definitions typically used:

```
#define obstack_chunk_alloc xmalloc
#define obstack_chunk_free free
```

See section 1.4.10 [Chunks], page 17, for full details.

Finally, when you link your program you must include among the arguments the file of obstack support functions, '`obstack.o`'. In the actual GNU system, this will no longer be necessary, because these subroutines will be in the C library.

1.4.2 Obstack Functions and Macros

The interfaces for using obstacks may be defined either as functions or as macros. If you are using an old-fashioned non-ANSI C compiler, they are actually defined only as macros. In ANSI C they are defined *both* as functions and as macros. We speak of them as *function/macros* and describe them like functions.

In non-ANSI C, the function/macro is really just a macro. Therefore, you can call it like a function, but you cannot use it in any other way (for example, you cannot take its address).

Calling the macros requires a special precaution: namely, the first operand (the obstack pointer) may not contain any side effects, because it may be computed more than once. For example, if you write this:

```
obstack_alloc (get_obstack (), 4);
```

you will find that `get_obstack` may be called several times. If you use `*obstack_list_ptr++` as the obstack pointer argument, you will get very strange results since the incrementation may occur several times.

In ANSI C, each function/macro has both a macro definition and a function definition. The function definition is used if you take the address of the function without calling it. An ordinary call uses the macro definition by default, but you can request the function definition instead by writing the function name in parentheses, as shown here:

```
char *x;
void *(*funcp) ();
/* Use the macro */
x = (char *) obstack_alloc (obptr, size);
/* Call the function */
x = (char *) (obstack_alloc) (obptr, size);
/* Take the address of the function */
funcp = obstack_alloc;
```

This is the same situation that applies for in ANSI C for the standard library functions.

If you do use the macros, even in ANSI C you must observe the precaution of avoiding side effects in the first operand, unless you are using the GNU C compiler. In that case, extra features of GNU C are used to define the macros so as to avoid any problem.

1.4.3 Obstack Data Structure

An obstack is represented by a data structure of type `struct obstack`. You can declare variables of this type and use them as obstacks, or you can allocate obstacks dynamically like any other kind of object.

In order to prepare the obstack structure for use in allocation, you must call the macro `obstack_init`. This is available only as a macro, not as a function, because a function definition would make it impossible to define `obstack_chunk_alloc` as a macro.

If `obstack_init` were a function, its declaration in ANSI C would look like this:

```
void obstack_init (struct obstack *obstack);
```

Here are two examples of how to allocate the space for an obstack and initialize it. First, an obstack that is a static variable:

```
struct obstack myobstack;
...
obstack_init (&myobstack);
```

and now an obstack that is itself dynamically allocated:

```
struct obstack *myobstack_ptr
```

```
        = ((struct obstack *)
           xmalloc (sizeof (struct obstack)));

obstack_init (myobstack_ptr);
```

1.4.4 Allocation in an Obstack

The most direct way to allocate an object in an obstack is with `obstack_alloc`, which is invoked almost like `malloc`. The ANSI C function declaration of `obstack_alloc` is as follows:

```
void * obstack_alloc (struct obstack *obstack_ptr, int size);
```

This function/macro allocates a block of *size* bytes in the obstack *obstack_ptr* and returns the address of the block. The contents of the block are not initialized.

To allocate a block with specified contents, use the function/macro `obstack_copy`, declared like this:

```
void * obstack_copy (struct obstack *obstack_ptr,
                     void *address, int size);
```

This initializes the newly allocated block with a copy of the *size* bytes of data starting at *address*.

The function/macro `obstack_copy0` is similar; the difference is that `obstack_copy0` appends an extra byte containing zero. This extra byte is not counted in the argument *size*. This function is convenient for copying a sequence of characters into an obstack as a null-terminated string. Here is an example of its use:

```
char *
obstack_savestring (char *addr, int size)
{
  return obstack_copy0 (&myobstack, addr, size);
}
```

Contrast this with the previous example of `savestring` using `malloc` (see section 1.3.1 [Allocation], page 2).

1.4.5 Freeing Objects in an Obstack

To free an object allocated in an obstack, use the function/macro `obstack_free`, whose ANSI C function declaration looks like this:

```
void obstack_free (struct obstack *obstack_ptr, void *varobject);
```

If *object* is zero, everything allocated in the obstack is freed. Otherwise, *object* must be the address of an object allocated in the obstack. Then *object* is freed, and so is every object that was allocated in the same obstack more recently than *object*.

When an entire chunk of memory in the obstack becomes free, the obstack package frees the chunk by calling your `obstack_chunk_free` function (see section 1.4.1 [Preparations], page 9). Then other obstacks, or non-obstack allocation, can reuse the chunk.

1.4.6 Growing Objects

Because storage in obstack chunks is used sequentially, it is possible to build up an object step by step, adding one or more bytes at a time to the end of the object. With this technique, you do not need to know how much data you will put in the object until you come to the end of it. We call this the technique of *growing objects*. The special function/macros for adding data to the growing object are described in this section.

You don't need to do anything special when you start to grow an object. Using one of the function/macros to add data to the object automatically starts it. However, it is necessary to say explicitly when the object is finished. This is done with the function/macro `obstack_finish`.

The actual address of the object thus built up is not known until the object is finished. Until then, it always remains possible that you will add so much data that the object must be copied into a new chunk.

While the obstack is in use for a growing object, you cannot use it for ordinary allocation of another object. If you try to do so, the space already added to the growing object will become part of the other object.

The most basic function/macro for adding to a growing object is `obstack_blank`, which adds space without initializing it. Here is its ANSI C function declaration:

```
void obstack_blank (struct obstack *obstack_ptr, int size);
```

To add a block of initialized space, use `obstack_grow`, which is the growing-object analogue of `obstack_copy`:

```
void obstack_grow (struct obstack *obstack_ptr,
                   char *data, int size);
```

Here *size* is the number of bytes to add, and *data* points to the data to initialize them with.

There is also a growing-object analogue of `obstack_copy0`, called `obstack_grow0`. It adds an additional zero-byte after the *size* specified bytes.

To add one character at a time, use `obstack_1grow`, whose function is declared like this:

```
void obstack_1grow (struct obstack *obstack_ptr,
                    char data_char);
```

A single byte containing *data_char* is added to the growing object.

When you are finished growing the object, use the function/macro `obstack_finish` to close it off and obtain its final address. The ANSI C function declaration for `obstack_finish` looks like this:

```
void *obstack_finish (struct obstack *obstack_ptr);
```

Once you have finished the object, the obstack is available for ordinary allocation or for growing another object.

When you build an object by growing it, you will probably need to know afterward how long it became. You need not keep track of this as you grow the object, because you can find out the length from the obstack just before finishing the object with the function/macro `obstack_object_size`, declared as follows:

```
int obstack_object_size (struct obstack *obstack_ptr);
```

Remember to do this *before* finishing the object. Once you finish the object, `obstack_object_size` will return zero.

If you have started growing an object and wish to cancel it, you should finish it and then free it, like this:

```
obstack_free (obstack_ptr, obstack_finish (obstack_ptr));
```

This has no effect if no object was growing.

1.4.7 Extra Fast Growing Objects

The usual function/macros for growing objects incur overhead for checking whether there is room for the new growth in the current chunk. If you are frequently construct objects in small steps of growth, this overhead can be significant.

You can reduce the overhead by using special "fast growth" function/macros that grow the object without checking. In order to have a robust program, you must do the checking yourself. If you do this checking in the simplest way each time you are about to add data to the object, you have not saved anything, because that is what the ordinary growth function/macros do. But if you can arrange to check less often, or check more efficiently, then the program runs faster.

The function/macro `obstack_room` returns the amount of room available in the current chunk. It is declared in ANSI C as follows:

```
int obstack_room (struct obstack *obstack_ptr);
```

This returns the number of bytes that can be added safely to the current growing object (or to an object about to be started) in obstack *obstack* using the super-fast methods.

While you know there is room, you can use these fast growth function/macros for adding data to a growing object:

```
void obstack_1grow_fast (struct obstack *obstack_ptr,
                         char data_char);
void obstack_blank_fast (struct obstack *obstack_ptr,
                         int size);
```

The function/macro `obstack_1grow_fast` adds one byte containing the character *data_char*; `obstack_blank_fast` adds a specified number of bytes without initializing them.

When the check using `obstack_room` fails, meaning that there is not enough room for what you want to add, the fast growth function/macros are not safe. Simply use the corresponding ordinary growth function/macro instead. Very soon this will copy the object to a new chunk; then there will be lots of room available again. The next time you check the available space, you will automatically go back to using the fast growth function/macros.

1.4.8 Status of an Obstack

Here are function/macros that provide information on the current status of allocation in an obstack. You can use them to learn about an object while still growing it.

`obstack_base` (*obstack*)

Returns the tentative address of the beginning of the currently growing object in *obstack*. If you finish the object immediately, it will have that address. If you make it larger first, it may outgrow the current chunk—then its address will change!

If no object is growing, this value says where the next object you allocate will start (once again assuming it fits in the current chunk).

Here is the function declaration in ANSI C:

```
void * obstack_base (struct obstack *obstack);
```

`obstack_next_free` (*obstack*)

Returns the address of the first free byte in the current chunk of *obstack*. This is the end of the currently growing object If no object is growing, `obstack_next_free` returns the same value as `obstack_base`.

Here is the function declaration in ANSI C:

```
void * obstack_next_free (struct obstack *obstack);
```

`obstack_object_size` (*obstack*)

Returns the size in bytes of the currently growing object. This is equivalent to

```
obstack_next_free (obstack) - obstack_base (obstack)
```

Here is the function declaration in ANSI C:

```
int obstack_object_size (struct obstack *obstack);
```

1.4.9 Alignment of Data in Obstacks

Each obstack has an *alignment boundary*; each object allocated in the obstack automatically starts on an address that is a multiple of the specified boundary. By default, this boundary is 4 bytes.

To access an obstack's alignment boundary, use the function/macro `obstack_alignment_mask`, whose ANSI C function declaration looks like this:

```
int obstack_alignment_mask (struct obstack *obstack_ptr);
```

The value is a bit mask; a bit that is 1 indicates that the corresponding bit in the address of an object should be 0. The mask value should be one less than a power of 2; the effect is that all object addresses are multiples of that power of 2. The default value of the mask is 3, so that addresses are multiples of 4. A mask value of 0 means an object can start on any multiple of 1 (that is, no alignment is required).

The expansion of the macro `obstack_alignment_mask` is an lvalue, so you can alter the mask by assignment. For example, this statement:

```
obstack_alignment_mask (obstack_ptr) = 0;
```

has the effect of turning off alignment processing in the specified obstack.

Note that a change in alignment mask does not take effect until *after* the next time an object is allocated or finished in the obstack. If you are not growing an object, you can make the new alignment mask take effect immediately by calling `obstack_finish`. This will finish a zero-length object and then do proper alignment for the next object.

1.4.10 Obstack Chunks

Obstacks work by allocating space for themselves in large chunks, and then parcelling out space in the chunks to satisfy your requests. Chunks are normally 4092 bytes long unless you specify a different chunk size. Regardless of the specified size, longer chunks will be allocated when necessary for long objects. The chunk size includes 8 bytes of overhead that are not actually used for storing objects.

The obstack package allocates chunks by calling the function `obstack_chunk_alloc`, which you must define. When a chunk is no longer needed because you have freed all the objects in it, the obstack package frees the chunk by calling `obstack_chunk_free`, which you must define as well.

These two must be defined (as macros) or declared (as functions) in each source file that uses `obstack_init` (see section 1.4.3 [Structure], page 11). Most often they are defined as macros like this:

```
#define obstack_chunk_alloc xmalloc
#define obstack_chunk_free free
```

Note that these are simple macros (no arguments). Macro definitions with arguments will not work! It is necessary that `obstack_chunk_alloc` or `obstack_chunk_free`, alone, expand into a function name if it is not itself a function name.

If they are macros, you must use only the macro definition of `obstack_init`, never the function definition.

The function that actually implements `obstack_chunk_alloc` cannot return "failure" in any fashion, because the obstack package is not prepared to handle failure. Therefore, `malloc` itself is not suitable. If the function cannot obtain space, it should either terminate the process or do a nonlocal exit using `longjmp`.

Because chunks are usually obtained with `malloc`, the chunk size should be 4 less than a power of 2. The default chunk size, 4092, was chosen because it fits this constraint, and it is long enough to satisfy many typical requests on the obstack yet short enough not to waste too much memory the portion of the last chunk not yet used.

The chunk size of an obstack with is the value of the function/macro `obstack_chunk_size`, whose ANSI C function declaration is thus:

```
int obstack_chunk_size (struct obstack *obstack_ptr);
```

Since this macro expands to an lvalue, you can specify a new chunk size by assigning it a new value. Doing so does not affect the chunks already allocated, but will change the size of chunk allocated for that particular obstack in the future. It is unlikely to be useful to make the chunk size smaller, but making it larger might improve efficiency if you are allocating many objects whose size is comparable to the chunk size. Here is how to do so cleanly:

```
if (obstack_chunk_size (obstack_ptr) < new_chunk_size)
  obstack_chunk_size (obstack_ptr) = new_chunk_size;
```

1.4.11 Obstacks and Signal Handling

Is it safe to use `obstack_alloc` in a signal handler? Yes, provided you are careful in the following ways:

- Use an obstack that you are certain the interrupted program is not trying to allocate or free in at the time the signal happens. (For example, you may have a special obstack for use in signal handlers.)
- Make sure that the `obstack_chunk_alloc` function is safe for use in signal handlers. If this function uses `malloc`, then you must use the GNU implementation of it (see section 1.3.8 [Malloc Signals], page 8).

The same consideration applies to any of the function/macros that allocate or grow objects.

It is not safe to use `obstack_free` in a signal handler, because this would require a safe `obstack_chunk_free` function, and that is currently not available. Normally this function is just `free`, which is not safe in signal handlers either in the GNU implementation or in Unix implementations.

1.4.12 Summary of Obstack Function/Macros

Here is a summary of all the function/macros associated with obstacks. Each takes the address of an obstack (`struct obstack *`) as its first argument.

`obstack_init` (*obstack_ptr*)
: Initialize use of an obstack. See section 1.4.3 [Structure], page 11.

`obstack_alloc` (*obstack_ptr, size*)
: Allocate an object of *size* uninitialized bytes. See section 1.4.4 [Obstack Allocation], page 12.

`obstack_copy` (*obstack_ptr, address, size*)
: Allocate an object of *size* bytes, with contents copied from *address*. See section 1.4.4 [Obstack Allocation], page 12.

`obstack_copy0` (*obstack_ptr, address, size*)
: Allocate an object of *size*+1 bytes, with *size* of them copied from *address*, and a zero-byte at the end. See section 1.4.4 [Obstack Allocation], page 12.

`obstack_free` (*obstack_ptr, object*)
: Free *object* (and everything allocated more recently than *object*). See section 1.4.5 [Obstack Freeing], page 13.

`obstack_blank` (*obstack_ptr, size*)
: Add *size* uninitialized bytes to a growing object. See section 1.4.6 [Growing], page 13.

`obstack_grow` (*obstack_ptr, address, size*)
: Add *size* bytes, copied from *address*, to a growing object. See section 1.4.6 [Growing], page 13.

`obstack_grow0 (`*obstack_ptr, address, size*`)`
: Add *size* bytes, copied from *address*, to a growing object, and then add another byte containing zero. See section 1.4.6 [Growing], page 13.

`obstack_1grow (`*obstack_ptr, data_char*`)`
: Add one byte containing *data_char* to a growing object. See section 1.4.6 [Growing], page 13.

`obstack_finish (`*obstack_ptr*`)`
: Finalize the object that is growing and return its permanent address. See section 1.4.6 [Growing], page 13.

`obstack_object_size (`*obstack_ptr*`)`
: Get the size of the currently growing object. See section 1.4.6 [Growing], page 13.

`obstack_blank_fast (`*obstack_ptr, size*`)`
: Add *size* uninitialized bytes to a growing object without checking that there is enough room. See section 1.4.7 [Fast Growing], page 15.

`obstack_1grow_fast (`*obstack_ptr, data_char*`)`
: Add one byte containing *data_char* to a growing object without checking that there is enough room. See section 1.4.7 [Fast Growing], page 15.

`obstack_room (`*obstack_ptr*`)`
: Get the amount of room now available for growing the current object. See section 1.4.7 [Fast Growing], page 15.

`obstack_alignment_mask (`*obstack_ptr*`)`
: The mask used for aligning the beginning of an object. This is an lvalue. See section 1.4.9 [Obstack Align], page 16.

`obstack_chunk_size (`*obstack_ptr*`)`
: The size for allocating chunks. This is an lvalue. See section 1.4.10 [Chunks], page 17.

`obstack_base (`*obstack_ptr*`)`
: Tentative starting address of the currently growing object. See section 1.4.8 [Status], page 16.

`obstack_next_free (`*obstack_ptr*`)`
: Address just after the end of the currently growing object. See section 1.4.8 [Status], page 16.

1.5 Automatic Storage with Variable Size

The function `alloca` supports a kind of half-dynamic allocation in which blocks are allocated dynamically but freed automatically.

Allocating a block with `alloca` is an explicit action; you can allocate as many blocks as you wish, and compute the size at run time. But all the blocks are freed when you exit the function that `alloca` was called from, just as if they were automatic variables declared in that function. There is no way to free the space explicitly.

The ANSI C declaration for `alloca` is as follows:

```
void * alloca (int size);
```

The return value of `alloca` is the address of a block of *size* bytes of storage, allocated in the stack frame of the calling function.

1.5.1 Example of `alloca`

As an example of use of `alloca`, here is a function that opens a file name made from concatenating two argument strings, and returns a file descriptor or minus one signifying failure:

```
int open2 (char *str1, char *str2, int flags, int mode)
{
  char *name = (char *) alloca (strlen (str1) + strlen (str2) + 1);
  strcpy (name, str1);
  strcat (name, str2);
  return open (name, flags, mode);
}
```

Here is how you would get the same results with `malloc` and `free`:

```
int open2 (char *str1, char *str2, int flags, int mode)
{
  char *name = (char *) malloc (strlen (str1) + strlen (str2) + 1);
  int desc;
  if (name == 0)
    fatal ("virtual memory exceeded");
  strcpy (name, str1);
  strcat (name, str2);
  desc = open (name, flags, mode);
  free (name);
  return desc;
}
```

As you can see, it is simpler with `alloca`. But `alloca` has other, more important advantages,

and some disadvantages.

1.5.2 Advantages of `alloca`

Here are the reasons why `alloca` may be preferable to `malloc`:

- Nonlocal exits done with `longjmp` automatically free the space allocated with `alloca` when they exit through the function that called `alloca`. This is the most important reason to use `alloca`.

  To illustrate this, suppose you have a function `open_or_report_error` which returns a descriptor, like `open`, if it succeeds, but does not return to its caller if it fails. If the file cannot be opened, it prints an error message and jumps out to the command level of your program using `longjmp`. Let's change `open2` to use this subroutine:

```
int open2 (char *str1, char *str2, int flags, int mode)
{
  char *name = (char *) alloca (strlen (str1) + strlen (str2) + 1);
  strcpy (name, str1);
  strcat (name, str2);
  return open_or_report_error (name, flags, mode);
}
```

  Because of the way `alloca` works, the storage for `name` is freed even when an error occurs, with no special effort required.

  By contrast, the other definition of `open2` (which uses `malloc` and `free`) would develop a storage leak if it were changed in this way. Even if you are willing to make more changes to fix it, there is no easy way to do so.

- Using `alloca` wastes very little space or time.
- Since `alloca` does not have separate pools for different sizes of block, space used for any size block can be reused for any other size.

1.5.3 Disadvantages of `alloca`

These are the disadvantages of `alloca` in comparison with `malloc`.

- If you try to allocate more storage than the machine can provide, you don't get a clean error message. Instead you get a fatal signal like the one you would get from an infinite recursion; probably a segmentation violation.
- Some non-GNU systems fail to support `alloca`, so it is less portable. However, a slower emulation of `alloca` written in C is available for use on systems with this deficiency.

1.5.4 GNU C Variable-Size Arrays

In GNU C, you can replace most use of `alloca` with an array of variable size. Here is how `open2` would look then:

```
int open2 (char *str1, char *str2, int flags, int mode)
{
  char name[strlen (str1) + strlen (str2) + 1];
  strcpy (name, str1);
  strcat (name, str2);
  return open (name, flags, mode);
}
```

But `alloca` is not always equivalent to a variable-sized array, for several reasons:

- A variable size array's space is freed on exit from the *block* it is declared in. The space allocated with `alloca` usually remains until the end of the function.
- It is possible to use `alloca` within a loop, allocating an additional block on each iteration. This is impossible with variable-sized arrays. On the other hand, this is also slightly unclean.

If you mix use of `alloca` and variable-sized arrays within one function, exiting a scope in which a variable-sized array was declared frees all blocks allocated with `alloca` during the execution of that scope.